IAP20 Rec'd PCT/PTO 13 JAN 2006

A Method of Decoding Variable Length Prefix Codes

BACKGROUND

1. FIELD

The present invention relates generally to decoding of variable-length prefix codes, e.g., Huffman codes, and, more specifically, to a new, combined decoding scheme of lookup table decoding and prefix oriented decoding.

2. DESCRIPTION

Entropy coding is a widely used data compression technique that many video and audio coding standards are based on. The theoretical basis of entropy coding states that a compression effect can be reached when the most frequently used data are coded with a fewer number of bits than the number of bits denoting the less frequently appearing data. This approach results in coded data streams composed of codes having different lengths.

There are a number of methods to form such variable length codes (VLC). One popular method uses a prefixed coding in which a code consists of a prefix that allows a decoding system to distinguish between different codes, and several significant bits representing a particular value (e.g., Huffman coding).

While most coding standards employ Huffman codes with prefixes composed of a series of '1' or '0' bits in their coding schemes, some standards (e.g., ISO/IEC 14496-2, Moving Pictures Experts Group (MPEG)-4 coding standard, Visual) allow for different coding schemes prefixed with a series of longer bit patterns.

As a general rule, the number of bits that comprise a variable length code depends on the number of bits that comprise the prefix of the code. At the same time, an experimentally defined subset of most frequently appearing codes may have relatively short prefixes (including zero prefix) and, thus, may be decoded in a lookup manner as a single code, which may be a faster way of decoding for a particular system.

Therefore, a need exists for the capability to provide high speed decoding of variable length codes prefixed with regular combinations of bits, in accordance with the actual frequency-to-code length distribution.

BRIEF DESCRIPTION OF THE DRAWINGS

The features and advantages of the present invention will become apparent from the following detailed description of the present invention in which:

Figure 1 is a diagram illustrating an exemplary variable length coding;

Figure 2 is a diagram illustrating relations between bits initially read from a bit stream, selected bits, and a table containing a decoded value, a validity indicator and auxiliary information; and

Figure 3 is a flow diagram illustrating the variable length decoding process in accordance with an embodiment of the present invention.

DETAILED DESCRIPTION

An embodiment of the present invention is a method of implementing a decoder for variable length codes that have prefixes composed of regular bit patterns. To apply the disclosed method to a particular coding scheme, such a scheme should comprise a subset of most frequently used codes with relatively short prefixes (including zero prefix), such that the prefix scan operation becomes inefficient. According to the disclosed method, the number of bits, not less than the maximal possible length of a VLC, is read from a bit stream. Then a predetermined number of bits is selected and used as an index to a data structure that contains at least a decoded value and validity indicator, along with other pre-decoded data, including but not limited to: prefix type and length, maximal code length for a group of codes, actual code length, and the number of bits to return to the bit stream. The validity indicator is used to determine whether to proceed with the decoding operation, or obtain the valid decoded value from the data structure and return excess bits to the bit stream. If the decoded value is indicated to be invalid, the decoding operation is continued, and a decoding method that estimates the length of the code prefix and the number of significant bits corresponding to the length estimated is applied to the bits initially read from the bit stream. The disclosed method requires less memory than direct lookup decoding methods, and performance of the method exhibits less memory access overhead as compared to prior art methods using multiple lookup tables. Additionally, the present

method appears to be more efficient for decoding of 'short prefix' codes as compared to other prefix oriented methods because it excludes operations of prefix type and length determination for the most frequently used codes.

Reference in the specification to "one embodiment" or "an embodiment" of the present invention means that a particular feature, structure or characteristic described in connection with the embodiment is included in at least one embodiment of the present invention. Thus, the appearances of the phrase "in one embodiment" appearing in various places throughout the specification are not necessarily all referring to the same embodiment.

Figure 1 is a diagram illustrating an exemplary variable length coding. As depicted by Figure 1, each variable length code has a group of bits used as a prefix 10 and a group of significant bits 12. The prefixes may be composed of a group of bits (bit patterns) that (in a general case) are replicated and concatenated to each other. The bits that follow the code prefix may be called significant bits.

Variable length codes (VLCs) may have identical prefixes. In this case, the codes constitute a prefix code group, but at the same time the number of significant bits that follow the prefix may differ. The maximal number of significant bits that is possible for a code in such a group may be referred to as the maximal bit number. The number of bits that follow the prefix for each VLC may be called the actual bit number.

Figure 2 is a diagram illustrating relations between bits initially read from a bit stream, selected bits, and a table containing a decoded value, a validity indicator and auxiliary information in accordance with an embodiment of the present invention. As depicted in the example of Figure 2, the number of bits 20 not less than any possible VLC length, i.e., the number of bits enough to contain the longest VLC in a particular coding scheme, may be read from a bit stream. Any number of leading bits 22 may be selected from the bits read. A data structure 24 is provided to contain at least decoded data and a validity indicator for each bit combination that may be formed from the selected bits. The data structure 24 may also contain auxiliary information on the type of prefix, code length, and the number of bits to return to the bit stream, in order to facilitate future decoding.

Figure 3 is a flow diagram illustrating a variable length decoding process in accordance with an embodiment of the present invention. At block 100, the number of bits not less than any possible variable length code is read from a bit stream. The number of bits read should be sufficient to contain the longest variable length code but is not limited

to store extra bits as it may facilitate the decoding process (e.g., the bits read fit the machine word size). Then, at block 102, the predetermined number of bits may be selected from the bits previously read. The number of bits to select depends on a particular coding scheme used, and, therefore, is determined by external means. The determination should be performed in a manner that allows the selected bits to span the most frequently used (the most probable) VLCs and at the same time to minimize the size of a code lookup table. At block 104 the code lookup table is indexed with the value formed from the selected bits, and at least a decoded value and a validity indicator, as well as auxiliary information are obtained. In one embodiment, obtaining the auxiliary information may be optional. The validity indicator is then checked at block 106, and if it is indicated to be valid, the decoded value obtained at block 104 is returned as the result of the decoding process at block 108. If necessary, the actual code length or the difference between the actual length and the number of selected bits (retrieved as auxiliary information at block 104) may be checked in order to adjust the bit stream after decoding.

If the decoded data is indicated to be invalid, a prefix oriented decoding method (i.e., a method that estimates the length of the code prefix and the number of significant bits corresponding to the length estimated) is applied at block 110 to the bits initially read from the bit stream. The auxiliary information obtained at block 104 may describe the type and length of the code prefix, and thus, increase the performance of the method to be further applied.

For an exemplary embodiment of the present invention implemented in the C and Assembler programming languages, refer to Appendix A. This example is non-limiting and one skilled in the art may implement the present invention in other programming languages without departing from the scope of the claimed invention.

The techniques described herein are not limited to any particular hardware or software configuration; they may find applicability in any computing or processing environment. The techniques may be implemented in logic embodied in hardware, software, or firmware components, or a combination of the above. The techniques may be implemented in programs executing on programmable machines such as mobile or stationary computers, personal digital assistants, set top boxes, cellular telephones and pagers, and other electronic devices, that each include a processor, a storage medium readable by the processor (including volatile and non-volatile memory and/or storage elements), at least one input device, and one or more output devices. Program code is applied to the data entered using the input device to perform the functions described and to

generate output information. The output information may be applied to one or more output devices. One of ordinary skill in the art may appreciate that the invention can be practiced with various computer system configurations, including multiprocessor systems, minicomputers, mainframe computers, and the like. The invention can also be practiced in distributed computing environments where tasks may be performed by remote processing devices that are linked through a communications network.

Each program may be implemented in a high level procedural or object oriented programming language to communicate with a processing system. However, programs may be implemented in assembly or machine language, if desired. In any case, the language may be compiled or interpreted.

Program instructions may be used to cause a general-purpose or special-purpose processing system that is programmed with the instructions to perform the operations described herein. Alternatively, the operations may be performed by specific hardware components that contain hardwired logic for performing the operations, or by any combination of programmed computer components and custom hardware components. The methods described herein may be provided as a computer program product that may include a machine readable medium having stored thereon instructions that may be used to program a processing system or other electronic device to perform the methods. The term "machine readable medium" used herein shall include any medium that is capable of storing or encoding a sequence of instructions for execution by the machine and that cause the machine to perform any one of the methods described herein. The term "machine readable medium" shall accordingly include, but not be limited to, solid-state memories, optical and magnetic disks, and a carrier wave that encodes a data signal. Furthermore, it is common in the art to speak of software, in one form or another (e.g., program, procedure, process, application, module, logic, and so on) as taking an action or causing a result. Such expressions are merely a shorthand way of stating the execution of the software by a processing system cause the processor to perform an action or produce a result.

While this invention has been described with reference to illustrative embodiments, this description is not intended to be construed in a limiting sense. Various modifications of the illustrative embodiments, as well as other embodiments of the invention, which are apparent to persons skilled in the art to which the invention pertains are deemed to lie within the spirit and scope of the invention.

APPENDIX A

GetVLC function (Assembler)

InitTable function ("C")

Input table ("C") and initial Huffman table (text)

Bit stream structure ("C")

Initial Huffman code table

```
/*
Codes          Vector differences
1              0
010            1
011           -1
0010           2
0011          -2
00010          3
00011         -3
0000110        4
0000111       -4
00001010       5
00001011      -5
00001000       6
00001001      -6
00000110       7
00000111      -7
0000010110     8
0000010111    -8
0000010100     9
0000010101    -9
0000010010    10
```

```
0000010011      -10
00000100010     11
00000100011     -1
00000100000     12
00000100001     -12
00000011110     13
00000011111     -13
00000011100     14
00000011101     -14
00000011010     15
00000011011     -15
00000011000     16
00000011001     -16
00000010110     17
00000010111     -17
00000010100     18
00000010101     -18
00000010010     19
00000010011     -19
00000010000     20
00000010001     -20
*/


---------------------------------------------
        Packed code/value table containing
information on prefix length and
significant bit number
---------------------------------------------
/// the table elements should be sorted by prefix length


static const long exTable_Mixed[] =
{
 13, /* max bits | bit-size flag */
```

```
7, /* number of prefix groups */
5, /* lookup table length (in bits) */

1, /* code length */
1, /* size of group */
0, /* bit index */
0, /* get bits */
0, /* unget bits */
0x00010000,
3, /* 3-bit codes */
2,
1,
1,
0,
0x00020001, 0x0003ffff,
4, /* 4-bit codes */
2,
2,
1,
0,
0x00020002, 0x0003fffe,
5, /* 5-bit codes */
2,
3,
1,
0,
0x00020003, 0x0003fffd,
8, /* 8-bit codes */
4,
4,
3,
0,
0x00080006, 0x0009fffa, 0x000a0005, 0x000bfffb,
```

```
7, /* 7-bit codes */
2,
4,
3,
1,
0x00060004, 0x0007fffc,
11, /* 11-bit codes */
4,
5,
5,
0,
0x0020000c, 0x0021fff4, 0x0022000b, 0x0023fff5,
10, /* 10-bit codes */
6,
5,
5,
1,
0x0012000a, 0x0013fff6, 0x00140009, 0x0015fff7, 0x00160008, 0x0017fff8,
8, /* 8-bit codes */
2,
5,
5,
3,
0x00060007, 0x0007fff9,
11, /* 11-bit codes */
16,
6,
4,
0,
0x00100014, 0x0011ffec, 0x00120013, 0x0013ffed, 0x00140012, 0x0015ffee,
0x00160011, 0x0017ffef, 0x00180010, 0x0019fff0, 0x001a000f, 0x001bfff1,
0x001c000e, 0x001dfff2, 0x001e000d, 0x001ffff3,
```

```
-1 /* end of table */
};

---------------------------
Bit Stream structures
---------------------------
typedef struct _MplDataBuf
{
   unsigned char *data;
   long        data_len;
   long        data_offset;
} MplDataBuf;

typedef struct _MplBitStream
{
   long        bit_ptr;        // Buffer bit pointer (31-0)

   MplDataBuf   *data_buf;      // Pointer to data and its size

   unsigned long *start_data;    // Internal bitsream pointers
   unsigned long *end_data;
   unsigned long *current_data;

   FILE        *fd;            // Input or output file

   jmp_buf     exit_point;     // Exit point to handle incorrect vlc codes
} MplBitStream;


#define DATA_BUFFER_SIZE         1*1024*1024

unsigned long bit_mask[33] =
{
```

```
    0x00000000,
    0x00000001, 0x00000003, 0x00000007, 0x0000000f,
    0x0000001f, 0x0000003f, 0x0000007f, 0x000000ff,
    0x000001ff, 0x000003ff, 0x000007ff, 0x00000fff,
    0x00001fff, 0x00003fff, 0x00007fff, 0x0000ffff,
    0x0001ffff, 0x0003ffff, 0x0007ffff, 0x000fffff,
    0x001fffff, 0x003fffff, 0x007fffff, 0x00ffffff,
    0x01ffffff, 0x03ffffff, 0x07ffffff, 0x0fffffff,
    0x1fffffff, 0x3fffffff, 0x7fffffff, 0xffffffff
};


--------------------------------------------
Function to form internal VLC table
--------------------------------------------
typedef unsigned long VLCDecodeTable;

static VLCDecodeTable* CreateVLCDecodeTable_Mixed(const long *src_table,
VLCDecodeTable *table, long *table_size, long cyr_size)
{
    int vm4_vlc_code_mask, vm4_vlc_data_mask, vm4_vlc_shift;
    int offset;
    int i, j;
    int code_length;
    int group_size;
    int bit_index;
    int get_bits;
    int unget_bits;
    int group_count;
    int outidx;
    int group_offset;
    int lookup_length;
    int prefix_offset;
```

```
switch(*src_table++ & VM4_VLC_LEN_FLAG)
{
case VM4_VLC_20:
    vm4_vlc_code_mask = 0xfffff000;
    vm4_vlc_data_mask = 0x00000fff;
    vm4_vlc_shift     = 12;
    break;
case VM4_VLC_24:
    vm4_vlc_code_mask = 0xffffff00;
    vm4_vlc_data_mask = 0x000000ff;
    vm4_vlc_shift     = 8;
    break;
default:
    vm4_vlc_code_mask = 0xffff0000;
    vm4_vlc_data_mask = 0x0000ffff;
    vm4_vlc_shift     = 16;
    break;
}

offset        = *src_table++ * 2;
lookup_length = *src_table++;
prefix_offset = (1 << lookup_length) * 2 + 2;
offset       += prefix_offset;

memset(table,  0, offset * sizeof(VLCDecodeTable));
///memset(table, -1, prefix_offset * sizeof(VLCDecodeTable));

table[0] = 32 - lookup_length;  /// the bit count to shift right
table[1] = prefix_offset;

while(*src_table != -1)
{
    code_length = *src_table++;
```

```
group_size  = *src_table++;
bit_index   = *src_table++ * 2 + prefix_offset;
get_bits    = *src_table++;
unget_bits  = *src_table++;

if(!table[bit_index])
{
   table[bit_index]     = get_bits;
   table[bit_index + 1] = group_offset = offset;
}
for(i = 0, group_count = 0; i < group_size; i++)
{

   if(code_length < lookup_length)
   {
      for(j = 0; j < (1 << (lookup_length - code_length)); j++)
      {
         outidx = ((((((unsigned long int)(*src_table & vm4_vlc_code_mask))
                  >> vm4_vlc_shift) & bit_mask[code_length])
                  << (lookup_length - code_length)) + j) * 2;

         table[outidx + 2]     = /*lookup_length - */code_length;
         table[outidx + 2 + 1] = ((*src_table & vm4_vlc_data_mask) << (32 –
                           vm4_vlc_shift)) >> (32 - vm4_vlc_shift);
      }
   }
   else if(code_length == lookup_length)
   {
      outidx = ((((unsigned long int)(*src_table & vm4_vlc_code_mask))
             >> vm4_vlc_shift) & bit_mask[code_length]) * 2;

      table[outidx + 2]     = code_length;///0;
      table[outidx + 2 + 1] = ((*src_table & vm4_vlc_data_mask) << (32 –
```

```
                                vm4_vlc_shift)) >> (32 - vm4_vlc_shift);
        }

        if(!unget_bits)
        {
          outidx = ((((unsigned long int)(*src_table & vm4_vlc_code_mask))
                  >> vm4_vlc_shift) & bit_mask[get_bits]) * 2;

          table[group_offset + outidx]    = ((*src_table & vm4_vlc_data_mask) <<
                                          (32 - vm4_vlc_shift)) >> (32 –
                                          vm4_vlc_shift);
          table[group_offset + outidx + 1] = 0;
          group_count++;
          src_table++;
        }
        else
        {
          for(j = 0; j < (1 << unget_bits); j++)
          {
            outidx = ((((((unsigned long int)(*src_table & vm4_vlc_code_mask))
                    >> vm4_vlc_shift) & bit_mask[get_bits - unget_bits])
                    << unget_bits) + j) * 2;

            table[group_offset + outidx]    = ((*src_table & vm4_vlc_data_mask)
                                            << (32 - vm4_vlc_shift)) >> (32 –
                                            vm4_vlc_shift);
            table[group_offset + outidx + 1] = unget_bits;
            group_count++;
          }
          src_table++;
        }
      }
      offset += group_count * 2;
```

```
    }

    *table_size = offset;

    return (VLCDecodeTable*)table;
}
```

Function to decode VLC (Assembler)

```
.686
.xmm
xmmword textequ <qword>
mmword      textequ <qword>
.model FLAT

MplDataBuf  struc   4t
data        dd      ?
data_len    dd      ?
data_offset dd      ?
MplDataBuf  ends

MplBitStream struc  4t
bit_ptr     dd      ?       ;;; Buffer bit pointer (31-0)

data_buf    dd      ?       ;;; Pointer to data and its size

start_data  dd      ?       ;;; Internal bitsream pointers
end_data    dd      ?
current_data dd     ?

fd          dd      ?       ;;; Input or output file
```

```
exit_point      dd      ?       ;;; Exit point to handle incorrect vlc codes
MplBitStream ends

_TEXT           segment

        extrn   _longjmp:near

;;; unsigned long asmbsGetVLC_LookupBitSearch
;;;                     (MplBitStream *bsm, const VLCDecodeTable *vlcTable)

_asmbsGetVLC_LookupBitSearch    proc    near
sizeof_locals   equu    14h
ws      equ     esp + 04h

bsm     equ     dword ptr [eax + 04h]
table   equ     dword ptr [eax + 08h]

        mov     eax,esp
        sub     esp,sizeof_locals
        and     esp,0fffffff0h
        push    eax
        mov     [ws],esi
        mov     [ws + 04h],edi
        mov     [ws + 08h],ecx
        mov     [ws + 0ch],ebx
        mov     [ws + 10h],ebp
        mov     esi,bsm
        mov     edi,table

        mov     ecx,1fh
        sub     ecx,MplBitStream.bit_ptr[esi]
        mov     ebx,MplBitStream.current_data[esi]
```

```
        mov     eax,[ebx]
        mov     edx,[ebx + 4]
        shld    eax,edx,cl              ;;; eax = data

        test    eax,eax
        jz      error_code              ;;; this branch is supposed not to be taken

        ;;; look up several bits first
        mov     ecx,[edi]                       ;;; ecx == 32 - lookup_bits
        mov     edx,eax
        shr     edx,cl
        mov     ebp,[edi + edx * 8 + 8]         ;;; ebp == (un)get bits
        or      ebp,ebp
        jz      scan                            ;;; not taken
        mov     eax,[edi + edx * 8 + 0ch]       ;;; eax == decoded data
        mov     ebx,MplBitStream.bit_ptr[esi]
        sub     ebx,ebp
        js      negative_ptr                    ;;; not taken

        ;;; exit
        mov     MplBitStream.bit_ptr[esi],ebx
        mov     esi,[ws]
        mov     edi,[ws + 04h]
        mov     ecx,[ws + 08h]
        mov     ebx,[ws + 0ch]
        mov     ebp,[ws + 10h]
        mov     esp,[esp]
        ret

scan:
        bsr     ecx,eax                         ;;; ecx = index

        mov     ebx,[edi + 4]           ;;; ebx == prefix_offset
```

```
        add     ebx,62
        mov     ebp,31
        sub     ebx,ecx
        sub     ebx,ecx                         ;;; ebx = offset (of bit index group)
        sub     ebp,ecx                         ;;; ebp = (31 - index)
        mov     edx,[edi + ebx * 4]     ;;; edx = get_bits
        mov     ebx,[edi + ebx * 4 + 4];;; ebx = offset (of code value and unget bits)

        sub     ecx,edx
        shr     eax,cl
        and     eax,bit_mask[edx * 4] ;;; eax = data

        lea     ebx,[ebx * 4]
        lea     ebx,[ebx + eax * 8]
        mov     ecx,[edi + ebx + 4]     ;;; ecx = unget_bits
        mov     eax,[edi + ebx]         ;;; eax = data

        mov     ebx,MplBitStream.bit_ptr[esi]
        lea     edx,[edx + ebp + 1]
        add     ebx,ecx
        sub     ebx,edx

        js      negative_ptr            ;;; not taken

almost_exit:
        mov     MplBitStream.bit_ptr[esi],ebx

exit:
        mov     esi,[ws]
        mov     edi,[ws + 04h]
        mov     ecx,[ws + 08h]
        mov     ebx,[ws + 0ch]
        mov     ebp,[ws + 10h]
```

```
        mov     esp,[esp]
        ret

negative_ptr:
        add     ebx,20h
        add     MplBitStream.current_data[esi],04h
        jmp     almost_exit             ;;; taken

error_code:
        push    -1
        lea     edx,MplBitStream.exit_point[esi]
        push    edx
        call    _longjmp
        ;;; no return here
        int     00h

_asmbsGetVLC_LookupBitSearch    endp

_TEXT       ends

_DATA       segment

bit_mask    dd      00000000h
            dd      00000001h, 00000003h, 00000007h, 0000000fh
            dd      0000001fh, 0000003fh, 0000007fh, 000000ffh
            dd      000001ffh, 000003ffh, 000007ffh, 00000fffh
            dd      00001fffh, 00003fffh, 00007fffh, 0000ffffh
            dd      0001ffffh, 0003ffffh, 0007ffffh, 000fffffh
            dd      001fffffh, 003fffffh, 007fffffh, 00ffffffh
            dd      01ffffffh, 03ffffffh, 07ffffffh, 0fffffffh
            dd      1fffffffh, 3fffffffh, 7fffffffh, 0ffffffffh
_DATA       ends
        end
```